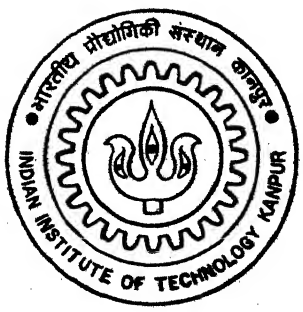
A MULTICAST TRANSPORT PROTOCOL

by

NAVUDU SRIN KUMAR



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1995

A MULTICAST TRANSPORT PROTOCOL

A thesis submitted

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Navudu Srin Kumar

to the

Department of Computer Science and Engineering

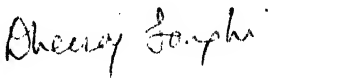
Indian Institute of Technology, Kanpur

January, 1995

CSE-1995-M-KUM-MUL

CERTIFICATE

It is certified that the work contained in the thesis entitled *A Multicast Transport Protocol*, by *N. Srin Kumar* has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.



Dr. Dheeraj Sanghi,
Assistant Professor,
Department of Computer Sc
and Engineering,
IIT Kanpur.

January, 1995

Abstract

New applications such as audio and video conferencing require multicast communication. Several schemes for multicasting have been proposed in literature, but all of them either require significant changes in the existing routers in the Internet, or do not provide sufficient flexibility to support different kinds of applications. In this thesis we designed a new multicast transport protocol, which can work on any network layer. The protocol provides varying levels of service quality, depending on the needs of an application. Hierarchical scheme is used in the protocol, to overcome the linear growth of protocol overhead, with increase in cardinality of the group. Multicast group consists of a leader, mini-leaders, and members. Leader is the principal member and present at the root, while mini-leaders are at the intermediate levels, and members at the bottom of hierarchy. We implemented this protocol as an application process using user datagram protocol. The unix platform used is sun-3 workstation, running SunOS 4.1.

Acknowledgement

I am grateful to my thesis supervisor Dr. Dheeraj Sanghi who initially suggested the interesting topic of Multicast Protocols for thesis work and gave me the opportunity to work with him. Without his constructive ideas, constant help and encouragement throughout the course of this work, I would still have been struggling with my embryonic concepts. Learning the style of technical writing from him has been one of the most joyful experiences during my M. Tech. program.

I wish to thank all of my classmates and friends for all the fun and enjoyment that we had together. In particular, I thank RS, Pavan, Parag, Brij, Shankar and Singhai. I would also like to thank the CSE staff for their co-operation and help.

Last but not the least, my sincere thanks go to my family whose care and love always inspired me to progress in life.

N. Srin Kumar

Contents

1 Introduction 1

1.1 Services Required 1

1.2 Message Ordering 2

1.3 Related Work 3

1.3.1 IP/Multicast 3

1.3.2 MTP 4

1.3.3 Internet Relay Chat Protocol 5

1.4 Need for a New Protocol 6

2 Protocol Design 7

2.1 Introduction 7

2.2 Design 8

2.3 Interface to User 9

2.4 Protocol States and Packets 11

2.5 Creating and Joining a Group 13

2.6 Quitting from a Group 14

2.7 Protocol Services 16

3 Implementation 19

3.1 System Design 19

3.2 Interface Between User and Protocol 22

3.3 Reliable Service 22

3.4 Testing 23

4 Conclusions **25**

4.1 Future Extensions 25

A Data Structures Used **30**

A.1 Data structures in Shared memory 30

A.1.1 Multicast control block (mcb). 30

A.1.2 Data structure for the Member. 32

A.2 Packet Formats 32

A.2.1 Join.Request Packet 32

A.2.2 Join.Accept Packet 32

A.2.3 Join.Deny Packet 32

A.2.4 M_cast & Forward Data Packets 33

A.2.5 M_ack & F_ack Packets 33

A.2.6 Quit & Cancel Request Packets 33

A.2.7 Quit.Confirm & Cancel.Ack Packets 33

B User Manual **34**

C Man Pages **38**

C.1 System Administrator's Manual 38

C.2 Man showmcb 40

List of Figures

2.1 Control Packet 12

2.2 Data Packet 12

2.3 Acknowledgment Packet 13

2.4 Creating and Joining a Group 14

2.5 Quitting From a Group 15

3.1 System Design 20

4.1 Topology of Multicast group 26

Chapter 1

Introduction

There is a growing demand for applications with multi-user interaction such as audio and video conferencing, distributed simulation, distribution of images, multi-user games, group communication, etc. These applications require development of one-to-many and many-to-many communication models as opposed to the traditional one-to-one communication model. A Multicast protocol plays a significant role in one-to-many and many-to-many communication models. A Multicast group is a collection of processes that are the recipients of a given set of messages. These messages may originate at one or more sites. A Multicast group defines host group, which is a set of hosts containing the current members of multicast group. A Multicast protocol at the network layer specifies the details of how hosts can send packets to a host group. A Multicast protocol at the transport layer specifies the details of how hosts can send packets to a multicast group.

1.1 Services Required

Depending on the application requirement, a multicast protocol should provide different classes of service. We have identified three classes of service. They are

Best-effort : This service is similar to the service provided by the Internet Protocol (IP) [16] in the usual one-to-one communication world. In this service, the packets may get lost, or may arrive at their destinations in an order different from the order in which they were sent. But unlike in IP, we would like the protocol to detect duplicate packets.

Intermediate : In this class of service, losses are tolerated, but order of packets should be correct, and duplicates are to be avoided.

Reliable : This service is similar to the service provided by Transmission Control Protocol (TCP) [17]. The multicast protocol would ensure that the recipients receive all the packets without any loss or duplicates and in the correct order.

In typical multicast applications, recipients may attach themselves to a group at any time, and leave at any time. Thus the notion of reliability is somewhat loose. It only means that all packets that are forwarded to an application must be in sequence, no duplicates, and no loss in that *range of packets.*

1.2 Message Ordering

In the context of multicast protocols, the concept of in-sequence delivery is different from that used in one-to-one communication. Two types of ordering of packets can be defined.

Uni-source ordering : In uni-source ordering, all packets from a single source should be delivered to all application processes in the same order as the sending order from the source. Packets from different sources can be interleaved in any order. In particular, overall order of packets at different destination may be different.

Multi-source ordering : In multi-source ordering, we insist that two packets should reach all recipients in the same relative order irrespective of whether the two packets come from the same source or different sources.

While uni-source ordering may suffice for most applications, multi-source ordering is required by some applications such as those in the banking sector. There are applications where the receipt of messages in different orders will lead to inconsistency. To illustrate this, consider a bank with two main computers in the main branch and each branch office having a computer connected to these main computers. The two main computers constitute a multicast group and each branch office is a potential source site. Each main computer has a copy of the entire banking database and will process all transactions arriving from branch offices. Whenever a branch office makes a transaction it is multicast to this group. The second main computer is needed for disaster recovery. The transactions should be

executed in the same order at the main computers like withdrawls and payments to avoid inconsistencies like overdraft penalties in one machine and no penalty in the other. This type of application is discussed in detail in Reference [6].

Another interesting area, where multi-source ordering is required, is multi-user games. For example, consider a war game, where two or more users can play the game from different sites (not necessarily different hosts). On each user site, the battle field is simulated. The snapshot of the battle field at the start of the game is the same to all users. Each user is provided with artillery, tanks, fighters, etc. A user can use his tanks and fighters to attack any other users tanks and fighters. Moves of each user are multicast to the group. In this case, to keep the battle field consistent at all sites, multi-source ordering of the moves is necessary.

1.3 Related Work

Various proposals for multicasting have been presented at different layers in the network architecture like network layer, transport layer, and application layer. Examples of network layer solutions are IP/Multicast [10], XTP [18, 7], and ST-II [21]. ST-II is a network layer, connection oriented, stream protocol. This supports guaranteed transfer as it reserves the resources along the path. This is a new protocol and needs the routing software present in the gateways to be changed if this protocol is to be used at its best. Currently this protocol this is not supported. MTP [1] is a transport layer protocol and IRC [15] (Internet Relay Chat) protocol is at the application layer. IP/Multicast, MTP, and IRC are worth discussing to understand the work done till now on multicasting.

1.3.1 IP/Multicast

In IP/Multicast protocol [10], a **host group** is defined as the set of hosts which are current members of the multicast group. A host group defines a **network group**, which is the set of networks containing current members of the host group. When a packet is sent to a host group, a copy is delivered to each network in the corresponding network group. Then, within each network, a copy is delivered to each host belonging to the group.

To support IP/Multicast delivery, every Internet gateway has to maintain the following data structures :

Routing Table : In addition to conventional internet routing information, the distance and direction to the nearest gateway on every network is stored.

Network Membership Table : It is a set of network membership records. Each record represents a host group and lists the networks to which members of the group are attached.

Local Host Membership Table : It is a set of records, one for each host group that has members on directly attached networks. Each local host membership record indicates the local hosts that are members of the associated host group.

Currently, a prototype (host group facility) is implemented as an extension of IP. For practical reasons, this prototype implements all group management functions and multicast routing outside of the Internet gateways in special hosts called multicast agents. The collection of multicast agents, in effect, provides a second gateway system on top of the existing Internet for multicast purposes. This causes redundancy of routing tables between gateways and multicast agents and the increased delay of extra hops in the delivery path.

1.3.2 MTP

MTP [1] is a transport protocol designed to support reliable multicast transmissions on top of IP/Multicast. It is based on the notion of a multicast master which controls all aspects of group communication.

Membership classes defined are consumer, producer and master. Each successive class is a formal superset of the previous. In MTP, a multicast group is a set of process and is referred to as a web. The master is the principal member of the web. The master is mainly responsible for giving out transmit tokens to members who wish to send data, and overseeing the web's membership and operational parameters. A producer is permitted to transmit data as well as receive data transmitted by other producers. A consumer is capable only of receiving user data. When a process joins the group, it specifies whether the receiver wishes to be a producer of information or only a receiver, whether the connection should be reliable or best-effort, whether the receiver is able to accept multiple senders of information, the minimum throughput desired, and the maximum data packet size. If the request can be granted, then the master replies with an ACK, otherwise a NAK is returned.

The master distributes transmit tokens to data producers in the group, which allow data to be provided at a specific rate. The master can simultaneously transmit 12 tokens but not more. Data is transmitted as messages. Each message can have any number of packets (network packets). End of message is indicated by voluntarily submitting the token to master. Each message is numbered by the master, and this number is used by all producers and consumers, to deliver messages in order to user.

MTP is a multi-source ordering protocol. Rate control provides flow control in the protocol. Members that are unable to maintain a minimum flow, are rejected. Error recovery utilizes a NAK-based selective retransmissions scheme. Senders are required to maintain data for a time-period specified by the master, and retransmit this data when requested by members of the group. If retransmission request arrives after the data has been released, the sender sends a NAK in response to the request.

1.3.3 Internet Relay Chat Protocol

The IRC [15] (Internet Relay Chat) protocol has been designed, for use with text based conferencing. A typical setup involves a single process (the server) forming a central point for clients (or other servers) to connect to, performing the required message delivery, multiplexing and other functions. Servers connected to each other form the backbone of the IRC network. Network configuration allowed for IRC servers is that of a spanning tree. A special class of clients called operators are defined and they perform general maintenance functions on the network such as disconnecting and reconnecting servers. Operators can close the connection between any client and server.

In IRC the multicast group is referred to as a channel. Two types of channels are allowed by this protocol. One is a distributed channel, which is known to all the servers that are connected to the network. The other one is a local channel. The local channel is known only to the server on which the channel is created. Due to this, only clients on the server where the local channel is created are allowed to join.

The IRC protocol has been implemented using Transmission Control Protocol (TCP) as the layer below. This protocol does not scale well when membership becomes very large. The main problem comes from the requirements that

- all servers should know about all the other servers and users. The information regarding the servers and users should be updated, as soon as it changes, and

- all servers should know about all channels, their inhabitants, and properties.

1.4 Need for a New Protocol

To serve the needs of applications we need a transport protocol that works on any network layer. The protocol should provide different classes of service to satisfy the need of applications. In the previous section, we summarized IP/Multicast, MTP, and IRC protocol. IP/Multicast does not have explicit set-up processing between the sender and the receiver prior to the establishment of group communication. Due to this, reliable service cannot be provided. In addition, for the IP/Multicast to be supported, changes have to be made in the existing IP layer software both in hosts and gateways. At present, only an experimental Mbone-structure is present in Internet, which supports these extensions. Due to this, the efficiency and the speed claimed by IP/Multicast is not achieved. MTP provides reliable service, but it uses negative acknowledgement scheme and needs network layer to support IP/Multicast.

Our goal is to provide a multicast transport protocol, so that, users need not design a new multicast protocol for every new application. For example, IRC protocol has to implement multicast protocol to suit its need. It did not use IP/Multicast or MTP because many hosts do not support multicasting at network layer. In general, any solution that needs large-scale changes in all gateways is not practical. So, until an effective network layer solution is agreed upon and implemented on a large scale, a transport layer solution seems attractive.

In the rest of this dissertation we describe a multicast protocol that works on any network layer. The implementation is done on top of UDP, but it could also be done on top of IP. This protocol supports the three different qualities of service; best-effort, intermediate, and reliable. This protocol has the concept of a leader, not unlike that in MTP [2]. The process which creates the group becomes the leader of the group. To decrease the control overhead on the leader, the protocol uses the hierarchical approach, and there can be leaders at different levels. (Leaders, other than the root, are called mini-leaders.) The leader distributes the messages to the multicast group, either directly, or through mini-leaders. Quality of service needed by an application is to be specified at the time of starting the group (which essentially means creating a leader).

Chapter 2

Protocol Design

2.1 Introduction

The goal of our effort is to develop a multicast protocol at the transport level. The protocol should assume that the underlying network does not support multicast addresses. (However, if parts of the network support multicasting, our protocol should be able to take advantage of that.) To serve the needs of several different applications, the multicast protocol should provide different qualities of service. We have identified three different qualities of services: best-effort, intermediate and reliable. In the best-effort service, duplicates are avoided, but losses and out of sequence packets may occur. In the intermediate service, duplicates and out of sequence packets are avoided, but it assumes that the application can tolerate losses. In reliable service, duplicates, losses and out of sequence packets are avoided.

The interface between the user and the protocol is specified by defining five operations. These operations are: create_group(), join_group(), quit_group(), send(), receive(). The protocol provides the service to the user through service access points(SAPs). SAPs are created by using any one of the two operations create_group or join_group. User can communicate with the protocol through theses SAPs, using operations Send and Receive. The SAP is closed by the operation quit_group. The number of service access points to be provided is left to the implementor. Each SAP is associated with a protocol entity. The requests made by the user through the SAP are satisfied by the protocol entity associated with it.

2.2 Design

A simple protocol to achieve multicast is to send a packet to each member of the group. In this protocol each member has to keep the information of all the remaining members. But, this leads to redundancy of information and multi-source ordering cannot be achieved. To overcome these problems, centralized control is used. In this scheme the *leader* (the first member of the group) has information about all the members of the group, while members only have the information about the leader. Whenever a member wants to multicast data, it sends the data to the leader and the leader in turn multicasts the data to all members. In this protocol, uni-source ordering is achieved, by having a sequence number in each packet sent by the member. Leader keeps track of the sequence numbers of all its members and receives the packets from the members in sequence. Leader replaces the sequence number in the packet by another sequence number, before sending it to the group. This new sequence number is common to all sources, and thus preserves the ordering across all sources. Members receive packets sent to the group and use this sequence number to preserve multi-source ordering.

To achieve multicast, the packet is sent by the leader to each member of the group. As the cardinality of the group increases, there is a linear increase in the protocol overhead. This is because the leader has to keep information about all its members. To overcome this problem of linear growth of protocol overhead, a hierarchical scheme is used. The number of members to which the leader can send data directly is restricted to some constant (say, 10). If the leader receives any further request to join the group, it is rejected. However, the members, themselves can become mini-leaders, and accept connections. In this fashion, the multicast group is organized in a tree-like hierarchy. The user has to select a mini-leader to whom it wants to connect. The cardinality of the mini-leaders is also restricted to the same constant. For most applications, 3-level hierarchy should be sufficient, though the protocol does not limit the number of levels in the multicast tree. (A complete tree with three levels of hierarchy, and 10 as the maximum cardinality of leader, will have 1111 members.) This hierarchical approach would be even more advantageous, if physically close hosts choose a common leader. In the rest of the chapter the word level refers to the level of this hierarchy. We consider the leader to be at the top level and members at the bottom level. Mini-leaders are present in levels between the leader and members. Multicast data refers to the packets

flowing from lower level to higher level (i.e., from member through mini-leaders to leader or, from mini-leader to leader). Forward data refers the packets flowing from higher level to lower levels (i.e., from leader through mini-leaders to member).

A multicast group consists of a leader, mini-leaders and members. Whenever a member wants to multicast data, it sends the packet to the mini-leader at the next higher level with a sequence number. Mini-leader keeps track of these sequence numbers of all its members and receives the packets from the member in sequence. Mini-leader replaces the sequence number in the packet, by another sequence number, before sending it to the next higher level. This new sequence number used by the mini-leader is from the common sequence space used for all its members and its multicast data. Thus the mini-leaders at all levels preserve the uni-source ordering. When leader receives multicast packets, it receives the packets in sequence from its member. Leader before sending it to all members, replaces the sequence number in the packet by another sequence number from the sequence space, which is common for all its members and its data. But, all the members and the mini-leaders keep track of this sequence number (sequence space) to receive the forward data. Mini-leaders when receive the forward packets, it sends a copy to the user and a copy each to the members with out changing the sequence number. As all members keep track of this sequence number and they receive the packets in sequence the multi-source ordering is preserved.

Each group is identified with a unique address, which is the leader's Internet address and a name (string of characters). The name used must be unique at that host. We refer to this pair as multicast group identifier. As the Internet address of a machine is unique in the network, and the group name is unique at the leader, the multicast group identifier becomes unique in the network.

2.3 Interface to User

Interface between the user and the protocol is given by five operations.

create_group: Parameters necessary for this operation are group_name, service and min_memb. Group_name refers to the name of the group to be created. Service refers to the type of service that the user is requesting from the protocol. Min_memb refers to the minimum number of members required before any data is transmitted to the group.

If the min_memb is greater than zero then the create_group operation gets blocked till the number of members in the group reaches min_memb. Create_group operation opens a SAP if one is available and returns the SAP number after creating the group. If a protocol entity in FREE state is not available then create_group operation returns an error. The protocol entity associated with this SAP becomes the leader of the group.

join_group: Parameters necessary for this operation are mid and mini_addr. Mid refers to the multicast group identifier which consists of the group name and the leader's internet address. Mini_addr refers to the mini-leader's internet address. This operation is invoked to become a member of an already existing group. Join_group operation opens a SAP, if no SAP is available, the operation returns an error. After opening a SAP the host sends a request to the mini-leader(leader). Host sends the request to the mini-leader if the parameter mini_addr is non-zero else it sends the request to the leader. If the request is accepted then the operation returns the SAP number opened. Otherwise it closes the SAP and returns an error.

quit_group: Parameter necessary for this operation is the SAP number returned by the create_group or join_group operations. This operation is used to quit from the group. For this, the protocol entity closes the SAP. The SAP may be reused by the implementation, depending on the state of the protocol entity. SAP cannot be used by the implementation, until it's protocol entity returns to the FREE state. When this operation is used, if protocol entity, associated with the SAP is a member, the host sends a request to it's mini-leader, requesting to stop sending forward data to it, and waits for conformation. On confirmation from mini-leader, the state of protocol entity is changed to FREE. If the protocol entity is a mini-leader or leader, the SAP is closed, but the protocol entity is not changed to FREE state. The state of this protocol entity is changed to FREE, when all members dependent on it quits the group.

send: Parameters necessary for this operation are SAP number, data_addr and size. SAP number is the one returned by join_group or create_group operation. Data_addr is the address of data where the data resides to multicast. Size is the size of data to be multicast. This operation is used to send data to the group. After the data is multicast, the operation returns the size of data multicast.

receive: Parameters necessary for this operation are SAP number, data_addr and size. Data_addr is the address where the data received is to be placed. Size gives the maximum

size of data to receive. This operation is used to receive the data sent to the group. After data is received, the operation returns the size of data received.

2.4 Protocol States and Packets

Protocol entity can be in one of the five states : FREE, JOIN_REQUEST, ESTABLISHED, CLOSEWAIT, or CLOSED state. All the protocol entities at the starting are initialized to FREE state.

FREE state If the protocol entity is in FREE state, it implies that the SAP associated with this protocol entity can be opened by implementation when requested by user.

JOIN_REQUEST state If the protocol entity is in join_request state, it implies that the protocol entity is waiting for a Join.Confirm or Join.Deny packet from its mini-leader.

ESTABLISHED state The protocol entity in ESTABLISHED state is ready to provide the service to the user through the associated SAP.

CLOSEWAIT state Protocol entity in CLOSEWAIT state has its SAP closed. This protocol entity waits for all members depending on it to quit the group.

CLOSED state If the protocol entity is in CLOSED state, it implies that it is waiting for a *Quit.Confirm* packet from its mini-leader(leader).

Protocol uses nine types of packets : *Join.Request, Join.Accept, Join.Deny, Quit.Request, Quit.Confirm, Forward.Data, Multicast.Data, Forward.Ack* and *Multicast.Ack.* The fields necessary and relevant in each packet are different. The necessary fields in each packet is given in Appendix A. The nine packets are classified into three types control packets, data packets and ack packets. *Join.Request, Join.Accept,* and *Join.Deny* packets are control packets. *Forward.Data, Multicast.Data, Quit.Request,* and *Quit.Confirm* packets are data packets. *Forward.Ack* and *Multicast.Ack* packets are ack packets.

Figure. 2.1 gives the format of control packet. Version specifies the protocol version, it is 1 for this protocol implementation. Type specifies the packet type. It can be a *Join.Request, Join.Accept,* or *Join.Deny.* Service specifies the class of service provided by the implementation to the group. Error gives the error number, which specifies the reason, for denying the request of member. Group name is a 12 byte field, this restricts the name of

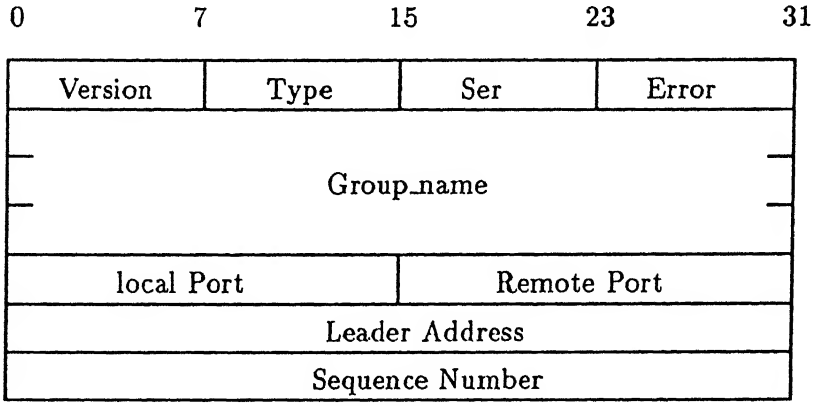


Figure 2.1: Control Packet

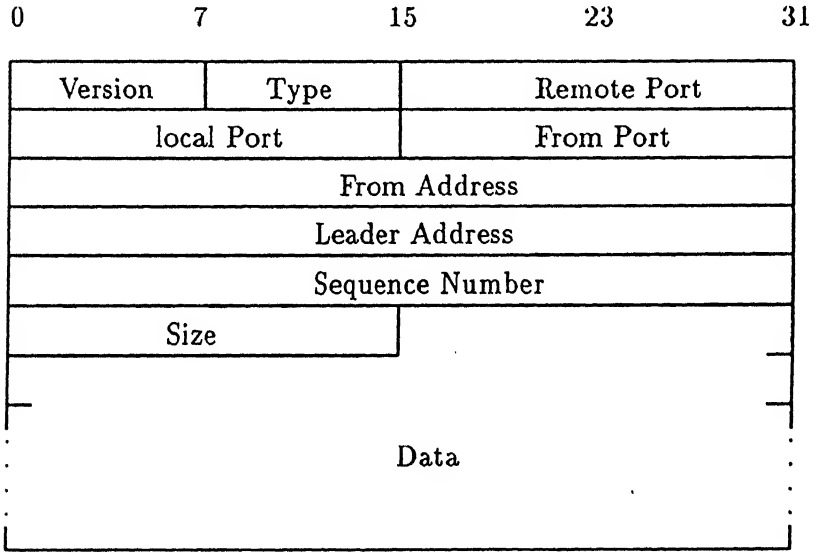


Figure 2.2: Data Packet

the group, maximum to 12 characters. Group name with leader address gives the multicast group identifier. Sequence number in control packet is used to convey the Initial Sequence number for the connection. Local port and Remote port are the source and destination port numbers of the packet.

Figure. 2.2 gives the format of data packet. Type can be *Multicast.Data, Forward.Data, Quit.Request,* or *Quit.Confirm.* From port gives the port number of the member who multicast the data. This field is set by the member multicasting the data to the group and this is not modified by mini-leader or leader. Similarly, from address gives the Internet address of the host, on which the member is and has multicast the data. Size gives the size of data in the packet.

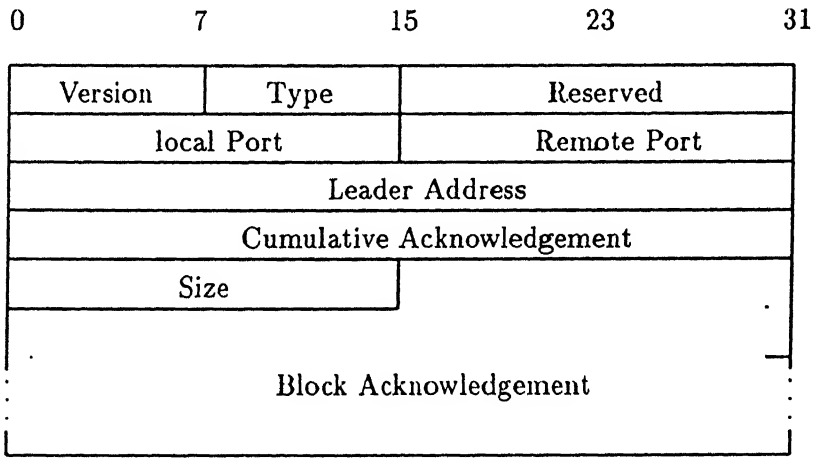


Figure 2.3: Acknowledgment Packet

Figure. 2.3 specifies the format of acknowledgement packet. Type can be *Multicast.Ack* or *Forward.Ack*. Size gives the size of block acknowledgement in the packet.

2.5 Creating and Joining a Group

State transactions while joining(creating) a group are given in Figure. 2.4. In the Figure for every state transaction, the condition required for state transaction and result of the state transaction are separated by a forward slash. When a process makes the create_group call, protocol searches for a protocol entity in FREE state and sets its state to ESTABLISHED state.

When a process makes the join_group call, protocol searches for a protocol entity in FREE state and sets its state to JOIN_REQUEST and blocks the join_group operation till it receives *Join.Accept* or *Join.Deny* packet. The SAP number and the Internet address of that host together form the member connection identifier. The protocol then sends a *Join.Request* packet to the specified mini-leader. *Join.Request* packet contains the member connection identifier, multicast group identifier, and the mini-leader's internet address.

When the *Join.Request* packet is received by the leader or mini-leader it sends the *Join.Accept* or *Join.Deny* packet to the member. If the specified mini-leader is not really a leader but an individual member, it becomes a mini-leader on receiving the *Join.Request* packet. It then accepts the connection by sending the *Join.Accept*. If the number of members at the mini-leader are already at maximum then, mini-leader sends a *Join.Deny* packet.

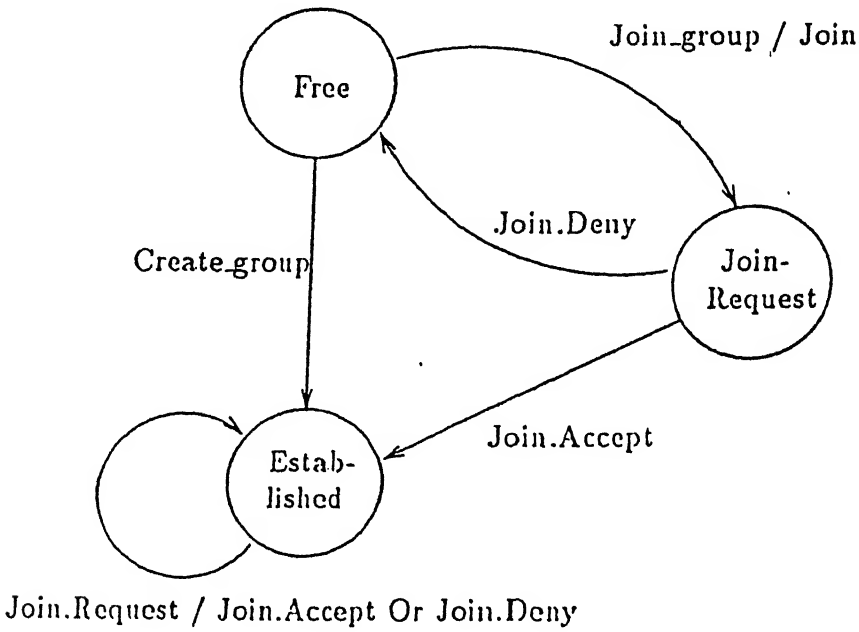


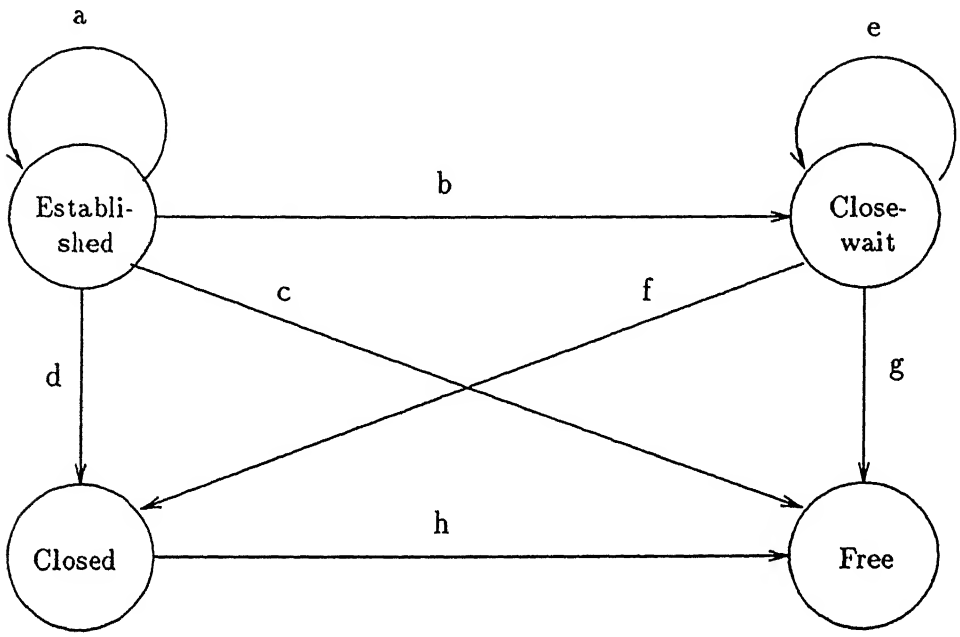
Figure 2.4: Creating and Joining a Group

When the leader (or mini-leader) sends the *Join.Accept* packet to the new member, it also informs about the quality of service provided on that group. The new member is added to the list of members, and any future message will also be sent to the new member. If the requesting host does not receive a *Join.Accept* or *Join.Deny* packet within a timeout period, the *Join.Request* packet is retransmitted.

When the protocol entity receives *Join.Accept* or *Join.Deny* packet it finds the respective SAP with the member connection identifier in the packet. If the SAP is in JOIN_REQUEST state and packet is *Join.Accept* then the protocol sets the state to ESTABLISHED and the join_group operation is released. If the state was JOIN_REQUEST and the packet Join.Deny then the host sets the state to FREE and releases the join_group operation by returning an error.

2.6 Quitting from a Group

State transactions while quitting the group are given in Figure. 2.5. Variable no_members keeps track of the number of members in the members list. Variable type specifies the type of the protocol entity (it can be a leader, mini-leader, or member). When a process makes a quit_group operation, the state of the protocol entity associated with the SAP is changed, and the SAP is closed. This operation is released after closing the SAP. If the protocol



a : Quit.Request / Quit.Confirm to member.

b : Quit_group & no_members != 0.

c : Quit_group & no_members = 0 & type = Leader.

d : Quit_group & no_members = 0 & type != Leader /
Quit.Request to mini-leader(leader).

e : Quit.Request & no_members != 1 / Quit.Confirm to member

f : Quit.Request & no_members = 1 & type != Leader /
Quit.Confirm to member & Quit.Request to mini-leader.

g : Quit.Request & no_members = 1 & type = Leader /
Quit.Confirm to member.

h : Quit.Confirm .

Figure 2.5: Quitting From a Group

entity is a member, the host sends a *Quit.Request* packet to its mini-leader(leader), changes its state to CLOSED and waits for the *Quit.Confirm* packet(sent by its mini-leader). If member does not receive a *Quit.Confirm* packet within a time period, it retransmits the *Quit.Request* packet to the mini-leader(leader). It is the responsibility of the member to confirm that it has quit from the group. If the protocol entity is a mini-leader, or leader with a non empty list of members, the state is changed from ESTABLISHED to CLOSEWAIT. If the protocol entity is a leader with an empty list of members, the state is changed from ESTABLISHED to FREE.

If a mini-leader or leader in ESTABLISHED state, receives a *Quit.Request* packet from its member, the member is removed from its list of members, and a *Quit.Confirm* packet is sent to the member. We mentioned in earlier section that if a member receives a *Join.Request* packet, then the member becomes a mini-leader and accepts the connection. Similarly, if the mini-leader receives a *Quit.Request* packet from member and it is the only member of the mini-leader, the mini-leader becomes a member and sends a *Quit.Confirm* packet.

If a mini-leader or leader receives *Quit.Request* packet from its member, in CLOSEWAIT state, and if the list of members is not empty after this member is removed, a *Quit.Confirm* packet is sent to the member. If a mini-leader in CLOSEWAIT state receives a *Quit.Request* packet and its list of members consist of only this member, then a *Quit.Confirm* packet is sent to its member, its state is changed to CLOSED, and a *Quit.Request* packet is sent to its mini-leader(leader). If a leader in CLOSEWAIT state receives a *Quit.Request* packet and its list forward members consist of only this member then a *Quit.Confirm* packet is sent to its member and state is changed to FREE.

In CLOSED state, if a protocol entity receives a *Quit.Confirm* packet, the state is changed to FREE.

2.7 Protocol Services

In best-effort service, the protocol avoids duplicates, but losses and out of sequence are not taken care of. To avoid duplicates the protocol has to remember the past packets it has sent to the user so that if a duplicate packet arrives the protocol can drop the packet. To remember all past packets, the protocol needs infinite memory which is not practical. To achieve this a variable `rnext` which is a sequence number is used. The protocol expects

the next packet to arrive have rnext as the sequence number . Status of a fixed number of packets whose sequence number follows rnext is stored. This number is to be decided by the implementor. Let this number be 'x' then the protocol stores the status of packets whose sequence number is between rnext and (rnext + x). The status of the packet is represented by a bit. If the bit is set it implies that the packet has arrived else it means that the packet has not arrived. In the rest of chapter, the variable flags represents the status of packets. In flags each bit from right to left represents the status of successive packets with sequence number after rnext (i.e. rnext + 1). Consider the following example.

rnext = 2356 and
flags = 0001 0000 0001 1100 0110 0000 0111 0011

indicate that packets with sequence numbers 2357, 2358, 2361 2362, 2363, 2370, 2371, 2375, 2376, 2377, 2385 have been received and the remaining ones have not yet been received. For updating flags and rnext the following four cases have to be considered.

case i. seq_no < rnext

In this case the packet is ignored considering the packet as duplicate.

case ii. seq_no = rnext

In this case the packet is the expected packet so rnext is to be updated to least seq_no not yet received for this all the 1's on the left hand side of the flag including the first zero are removed(shifted) and the number of digits shifted are added to rnext.

case iii. seq_no > rnext and seq_no ≤ rnext + 'x' In this case the status of the packet received is currently recorded in flags. First it has to be identified whether it is a duplicate or not. If the status of the packet is 1 then it is a duplicate. If the packet is not duplicate then the status of the packet is changed to 1.

case iv. seq_no > rnext + 'x'

In this case the status of the packet is not in the range that can be recorded so, to record the status of this packet the flags are to be shifted and rnext is to be updated such that this packet's status can be recorded in flags.

For this first the flags are shifted such that the seq_no -1 packet's status is indicated by 'x' bit and the first bit indicates (seq_no - x) packet. rnext indicates the least

`seq_no` packet that has not yet received. so for this the `flags` are shifted till the first left most zero is removed and accordingly the `rnext` is set.

In the intermediate service, the protocol avoids duplicates and out of sequence packets but assumes that the application can tolerate loses. The duplicates are avoided in the same way as in the best-effort service. When out of sequence packets are received they are buffered instead of delivering to user. After updating `rnext`, buffered packets whose sequence number is less than `rnext` are delivered in sequence to user.

In reliable service, to avoid losses positive acknowledgement scheme is used. If an acknowledgement for every packet is sent by all members, the overhead of acknowledgement processing becomes very large. To reduce the overhead, block acknowledgement (the size of the block is defined by implementation) is used. The acknowledgement packet consists of a sequence number called c_ack, which gives the cumulative acknowledgement and a series of bits representing the status of packets whose sequence number follows the c_ack sequence number. The number of bits should be a multiple of block_size (the exact number is to be decided by implementation). A timer for packet retransmission is set at each end of block transfer. When the timer goes off the protocol checks for the acknowledgements for that block only and if it does not receive the acknowledgements from all the members the packet is retransmitted. It is preferable to retransmit the packet selectively to those members who did not receive than sending it to all members.

Chapter 3

Implementation

We have implemented this protocol as an application process, using user datagram protocol(UDP). The Unix platform used is a SUN-3 workstation, running SunOS 4.1. The protocol could be implemented directly on top of IP (using raw IP sockets), but raw sockets can be accessed only by a super user. The user interface consists of five operations, each of which has been implemented by a C function. The five functions are create_group, join_group, quit_group, m_send, and m_receive. These functions' object code is placed in an archive called mlib.a. User has to link this archive with his code. The operations send and receive which are given in the protocol design are named here as m_send and m_receive to distinguish them from the standard send and receive system calls.

3.1 System Design

System level design is shown in Figure. 3.1. Five processes are used to implement the protocol. The processes are named as dem_process, recv_process, send_process, cont_process, and timer_process. Shared memory is used to store the multicast control blocks (mcb). Each mcb stores all relevant information of a particular connection. The fields of mcb structure are given in Appendix A. Processes and shared memory are represented as rectangles in the figure. The straight line connecting shared memory and the process shows the access of shared memory to the process. Shared memory is accessed by cont_process, send_process, and recv_process.

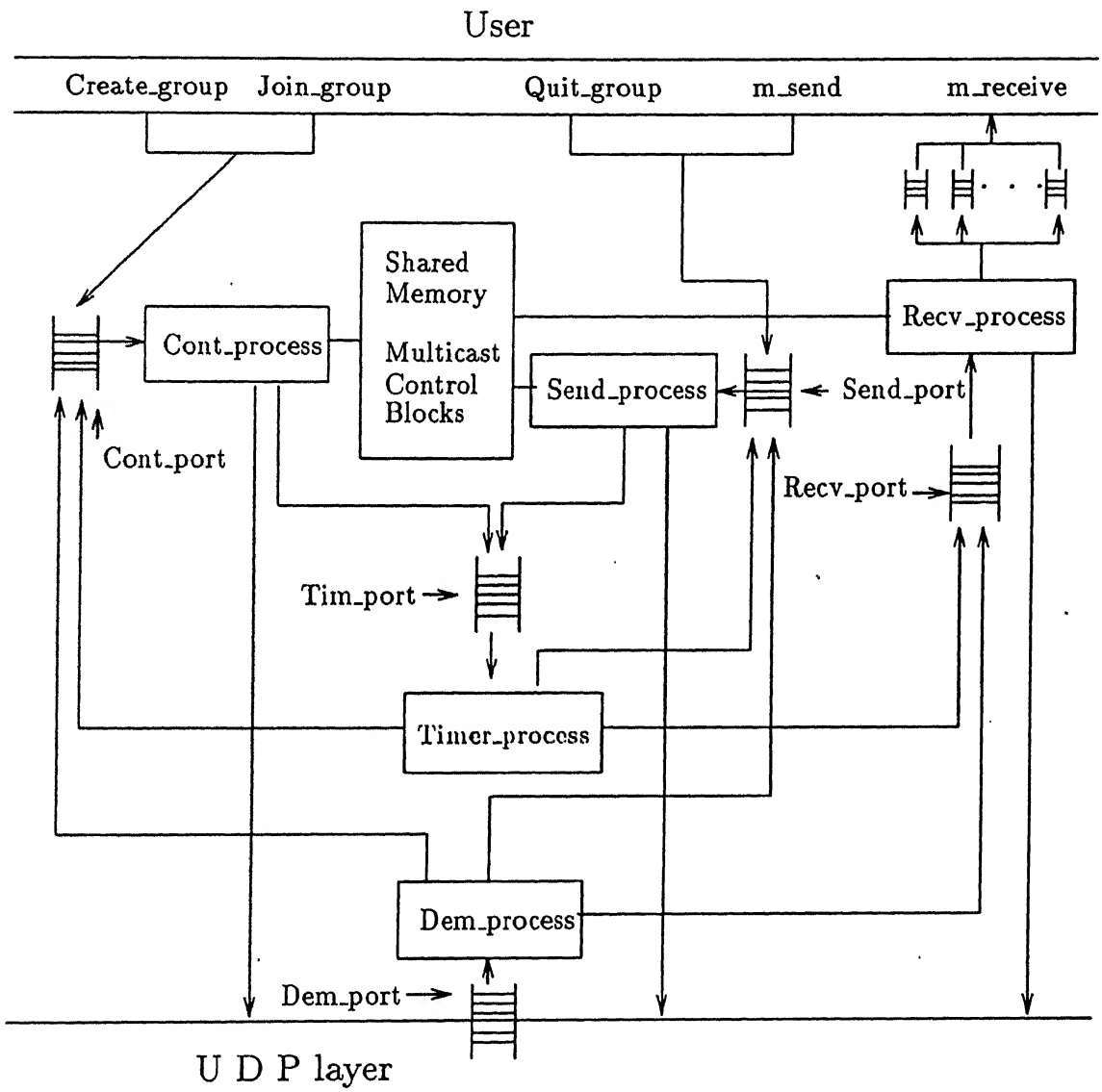


Figure 3.1: System Design

Dem_process demultiplexes the packets that are coming from the UDP layer through the port (DEM_PORT) and sends it to one of cont_process, recv_process, and send_process. These processes receive packets only through CONT_PORT, RECV_PORT and SEND_PORT. So, from now onwards, whenever we say that a packet is sent to a process, we mean that the packet is sent to the respective port. If a process sends packets to a port, it is indicated in the figure by an arrow pointing from the process to the port. The cont_process, controls the connection establishment and creation of the group. The send_process, multicasts data to the group, if it is a leader, or sends data to its mini-leader(leader), if it is a member. It also forwards the data, received from mini-leader(leader), to its members. This process stores the data till it gets the acknowledgement, in the case of reliable service. This process also takes care of connection closing. The recv_process receives the data from the UDP layer, forwarded by dem_process and demultiplexes it to the user processes, depending on the destination port in the packet. Recv_process also sends a copy of data to the send_process, if it is a mini-leader.

Send_process, cont_process, or recv_process sets the timer, by sending a packet to the TIMER_PORT. These packets have the information of, the process sending the packet, the time for which the timer has to be set, and the data to be sent on the expiry of timer. Timer_process receives packets through the TIMER_PORT and sets the timer for the requested time. When timeout occurs for a packet, the timer_process sends the data in the packet, back to the process, which had sent the packet. Port number DEM_PORT is fixed for all protocol implementations. This port number should be same on all the hosts communicating with each other. Through this port, multicast protocol messages are sent from one host to another. The port numbers SEND_PORT, RECV_PORT, CONT_PORT and TIMER_PORT can be different on different machines. Whenever a process wants to send a packet to other host, it sends directly to the DEM_PORT on that host. This is indicated in the figure with an arrow pointing from the process to the UDP layer.

In this implementation, Control packets (i.e., *Join.Request, Join.Accept,* and *Join.Deny* packets) are received by cont_process. *Multicast.Data, Forward.Ack, Multicast.Ack, Quit.Request,* and *Quit.Confirm* packets are received by send_process. *Forward.Data* packets are received by recv_process.

In this implementation we used packets in addition to the ones specified by protocol. This facilitates exchange of information between

- Send_process, recv_process, cont_process and timer_process.

Packets *Join.Group, Join.Group.Reply, Create.Group, Create.Group.Reply, Quit.Group, Quit.Group.Reply, User.Data,* and *User.Data.Reply* are used to exchange information between the user functions and protocol implementation. Send_process creates a message queue, through which it sends the *Quit.Group.Reply* and *User.Data.Reply* packets to user process. Packets *Re.Join.Request, Re.Quit.Request, Re.forward.Data,* and *Re.Mcast.Data* are used to exchange information between timer_process and other processes.

3.2 Interface Between User and Protocol

Interface between user and protocol is specified by five operations. These operations are create_group, join_group, quit_group, m_recv and m_send. They are implemented by C functions whose object code is put in an archive library. Create_group function sends *Create.Group* packet to cont_process and waits for *Create.Group.Reply* packet. On receiving the *Create.Group.Reply* packet the function returns. Similarly, join_group function sends *Join.Group* packet to cont_process and waits for *Join.Group.Reply* packet. Create_group and Join_group functions create a socket, and uses it, to send and receive their packets. These functions return a multicast identifier(mid). mid is an array of three integers, the first element is the socket identifier, second is the message queue identifier, and the third is message number. Socket identifier, is of the socket, they created. Message queue identifier is of the message queue created by send_process. message number is sent by send_process in *Create.Group.Reply* or *Join.Group.Reply* packets. This message number and message queue identifier are latter used by quit_group and m_send functions to receive the packets from send_process. Quit_group and m_send functions sends *Quit.Group* and *User.Data* packets respectively to send_process, and wait for *Quit.Group.Reply* and *User.Data.Reply* packets respectively. M_recv function receives data using the socket, created by create_group or join_group functions.

3.3 Reliable Service

In reliable service the protocol uses a block acknowledgement scheme. For this, generally, retransmission timer is set at end of each block transfer, and the end of block is indicated

to the receiver. Receiver on receiving the end of block, sends the acknowledgement. In this implementation, we avoided this by setting the timer for a packet, when its sequence number is a multiple of block size, and sending acknowledgement, on receiving a packet whose sequence number is a multiple of block size. Acknowledgement is also sent, when a duplicate packet is received. Send_process uses three queues `r_queue`, `m_queue`, and `f_queue` to store data, while providing reliable service. `R_queue` is used, to store the out of sequence packets, received from mini-leader(leader). `M_queue` is used to store the packets, which are multicast(i.e., sent to mini-leader(leader)), and waiting for acknowledgement. `F_queue` is used to store the packets, which are forwarded(i.e., sent to members), and waiting for acknowledgement. Send_process on receiving a *Multicast.Ack* packet, deletes the packets from the `m_queue`, which are acknowledged. Packet from `f_queue` should be deleted only after, all members have acknowledged it. For this we used a field in member structure called `acks`, and a field in member structure called `rack`(acknowledgement to be received). `Acks` is a vector of bits(32), each bit representing the status of acknowledgement, of a packet. The right most bit in `acks`, represent the status of acknowledgement, of the packet, whose sequence number is `rack`. In `acks`, the bits from right to left, represent the acknowledgement status of packets, whose sequence numbers are successive. On receiving a *Forward.Ack* packet from a member, `send_process` updates the `acks` field of that member, to represent the latest status. When timeout occurs for a *Forward.Data* packet, `send_process` checks the acknowledgement status of all members. If the packet is acknowledged by all members, the packet is removed from the `m_queue`, otherwise the packet is retransmitted. When a packet is removed from the `m_queue` the `rack` and `acks` are updated.

3.4 Testing

The implementation of the protocol is done on SunOS 4.1. For Testing this protocol, on network conditions where duplicates, out of sequences, and losses occur, we, needed to simulate that environment. These network conditions were simulated by using another demultiplexing process called tdem instead of dem_process. Dem_process demultiplexes the packets it receives from the DEM_PORT to cont_process, send_process and recv_process. But, tdem does not only demultiplexes but also simulate losses, out of sequences, and duplicates. Losses are simulated by dropping some packets. Out of sequences are simulated by storing a

packet and not sending it at its arrival time but, sending after another two or three packets are demultiplexed. Duplicates are simulated by sending a packet which has been copied and demultiplexed before.

In demt process, a counter is used to count the number of packets. This counter is a mod 4 counter, so, the value of it never exceeds 3. Using this counter, demt process simulates the network conditions necessary. If the counter is 0, the packet is copied into a buffer `buff1` and it is demultiplexed. If the counter is 1, the packet is copied into a buffer `buff2`, but, it's not demultiplexed. If counter is 2, the packet is demultiplexed. If counter is 3, the received packet is dropped simulating a loss, the packet in `buff1` is demultiplexed creating a duplicate, and the packet in `buff2` is demultiplexed creating an out of sequence packet.

Using this demt process the protocol was run and the services were found to give the satisfactory results.

Chapter 4

Conclusions

In this thesis we described a multicast transport protocol. The protocol can be implemented on top of a network layer which does not support multicasting. The protocol provides three levels of service quality, best-effort, intermediate, and reliable. In intermediate and reliable services protocol preserves multi-source ordering. Protocol provides five functions *create_group, join_group, quit_group, m_send* and *m_recv*, as an interface to the protocol. Multicast group is created dynamically with *create_group* function. This group ceases to exist, when all members quit from the group. We implemented this protocol as an application process (or rather, a set of processes), using UDP for inter-process communication. The implementation is done on a SUN-3, running SunOS 4.1 operating system.

4.1 Future Extensions

In our current design, we are fixing the quality of service for the group at the time of creating the group. So all the members of the group are forced to have the same service. An extension to this would be to allow the members to have a different quality of service than what had been fixed at the time of creating the group.

In our current design, protocol allows any user to join the group, if he knows the group name and internet address of the leader. Because of this, group is becoming open for all the users. An extension to this would be providing a closed group, in which the protocol allows only the members knowing the password in addition to the group name and internet address, to join the group.

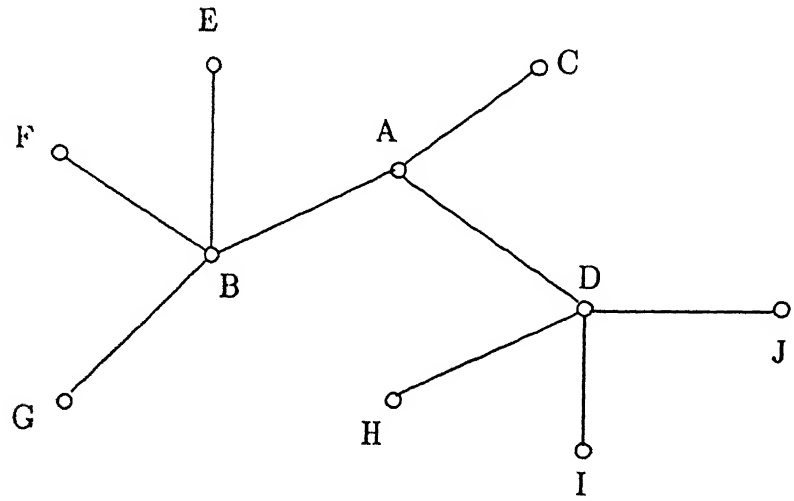


Figure 4.1: Topology of Multicast group

Protocol provides multi-source ordering for all the applications, even when the applications are in not need of it. An extension to this would be, to provide applications only uni-source ordering which do not need multi-source ordering. In our current design multicast group consists of always a leader, mini-leaders, and members. But, in a group which needs only uni-source ordering this hierarchy is unnecessary, it consists of only a flat graph with tree topology. In such a group, when a process in the group wants to multicast a packet, host sends that packet to all its neighbouring nodes in the graph. Node receiving a packet from its neighbouring node sends the packet to all its members other than the one from which it received. There would be no cycles in the graph of the nodes, because whenever a node joins the group it gets connected to only one node in the group.

Figure. 4.1 gives the topology of a group which provides only uni-source ordering. In this figure A, B, C, D, E, F, G, H, I, and J are members in the group. When F wants to multicast a packet, it sends the packet to its only neighbour node B. B on receiving the packet from F, sends the packet to E, G, and A. A on receiving the packet sends it to C and D. D on receiving the packet, sends it to H, I, and J. In this way the packet is sent to all members in the group. If A wants to multicast a packet it sends the packet to its neighbour nodes B, C, and D. B on receiving the packet sends it to E, F, and G. D on receiving the packet sends it to H, I, and J. Uni-source ordering is preserved in this method by each node using a sequence number in the packet, while sending it to its neighbour nodes, and keeping track of the sequence numbers of all its neighbour nodes.

Another interesting problem will be to create a directory service to advertise multicast groups, similar to *sd* in case of Internet MBone.

Bibliography

[1] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. Request for Comment: RFC-1301, Network Information Center, SRI International, February 1992.

[2] AT&T. *UNIX System V/386 Release 4 - Networks user's and Administrator's guide.* Prentice-Hall of India., 1990.

[3] AT&T. *UNIX System V/386 Release 4 - Programmers Guide and Networking Interfaces.* Prentice-Hall of India., 1990.

[4] P. Bhagwat, P. P. Mishra, and S. K. Tripathi. Effect of topology on performance of reliable multicast communication. Technical report, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, College Park, MD 20742, 1994.

[5] R. Braudes and S. Zabele. Requirements for multicast protocols. Request for Comment: RFC-1458, Network Information Center, SRI International, May 1993.

[6] J. Chang and N. F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[7] G. Chesson. XTP/PE overview. In *Proceedings of 13th Conference on Local Computer Networks*, pages 292–296, Minneapolis, MN, October 1988.

[8] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, volume 1. Prentice Hall, Englewodd Cliffs, NJ, 2nd edition, 1991.

[9] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: Design, Implementation, and Internals*, volume 2. Prentice Hall, Englewodd Cliffs, NJ, 1991.

[10] S. E. Deering and D. R. Cheriton. Host Groups: A multicast extension to the Internet Protocol. Request for Comment: RFC-966, Network Information Center, SRI International, December 1985.

[11] M. Gandhi, T. Shetty, and R.Shah. *Vijay Mukhi's The 'C' Odyssey UNIX - The Open, Boundless C.* BPB Publications, 1st edition, 1992.

[12] J. J. Gray and M. Anderton. Distributed computed systems. *Proceedings of the IEEE*, pages 719–726, May 1987.

[13] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, May 1989.

[14] H. Garcia Molina and M. Anderton. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[15] J. Oikarinen and D. Reed. Internet relay chat protocol. Request for Comment: RFC-1459, Network Information Center, SRI International, May 1993.

[16] J. B. Postel. Internet Protocol. Request for Comment: RFC-791, Network Information Center, SRI International, September 1981.

[17] J. B. Postel. Transmission Control Protocol. Request for Comment: RFC-793, Network Information Center, SRI International, September 1981.

[18] Robert M. Sanders and Alfred C. Weaver. The Xpress Transfer Protocol (XTP) — A Tutorial. *Computer Communication Review*, 20(5):67–80, October 1990.

[19] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[20] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[21] C. Topolcic. Experimental internet stream protocol version 2 (ST-II). Request for Comment: RFC-1190, Network Information Center, SRI International, October 1990.

Appendix A

Data Structures Used

A.1 Data structures in Shared memory

A.1.1 Multicast control block (mcb).

MAX_MEMBERS = maximum number of members at one level of hierarchy.

```
struct mcb_stru {
        unsigned char state ;                   /* state of the connection      */
        unsigned short no_members ;             /* number of members            */
        unsigned short min_members ;            /* minimum members required     */
        unsigned char type ;                    /* leader, mini or member       */
        char group_name[G_NAME_LEN + 1] ;       /* group name                   */
        unsigned char service ;                 /* type of service              */
        unsigned char echo ;                    /* echo the packets or not      */
        unsigned char m_flag ;                  /* flag indicates whether       */
                        /* packets are precent in any of the members            */
                        /* queue to multicast                                   */
        unsigned char f_flag ;                  /* flag indicates whether       */
                        /* packets are present in r_queue to forward            */
                        /* to the members                                       */
        unsigned char m_member ;                /* gives the index of the       */
```

```
                    /* member to check whether any packets are  */
                    /* there to multicast                       */
unsigned long leader_addr ;         /* leader's internet address    */
unsigned short local_port ;         /* local port number            */
unsigned long mini_addr ;           /* mini leader's internet addr  */
unsigned short mini_port ;          /* port number of mini leader   */
unsigned long snext ;               /* seq no. for the next packet  */
                                    /* to be sent to leader         */
unsigned long rnext ;               /* packet's sequence number     */
                    /* expected to receive next from mini leader    */
long pflags ;                       /* each bit from right to left  */
                    /* indicates the position of the successive     */
                    /* packets after rnext                          */
unsigned long rack ;                /* seq_no of the first packet in */
                    /* the f_queue waiting for acknowledgement      */
struct node *r_queue ;              /* list containing packets to   */
                                    /* be forwarded to the members  */
struct node *f_queue ;              /* list containing data forwarded */
                                    /* to members waiting for acks  */
struct node *m_queue ;              /* list containing data muticasted */
                                    /* to members waiting for acks  */
unsigned char no_packs ;            /* number of packet in r_queue  */
                                    /* waiting to be forwarded      */
unsigned char no_f_packs ;          /* number of packets in f_queue */
                                    /* waiting for acks             */
unsigned char no_m_packs ;          /* number of packets in multicast */
                                    /* queue waiting for acks       */
struct member_struct member[MAX_MEMBERS] ;
}
```

A.1.2 Data structure for the Member.

```
struct member_struct {
        unsigned short member_port ;      /* 16-bit port number        */
                                          /* network byte ordered      */
        unsigned short no_packets ;       /* number of packets in      */
                                          /* the queue                 */
        unsigned long member_addr ;       /* 32-bit netid/hostid       */
                                          /* network byte ordered      */
        unsigned long rseq ;              /* sequence no. next to      */
                                          /* receive from this mem     */
        unsigned long pflags ;            /* each bit from right to    */
                        /* left indicates the successive packet        */
                        /* positions after rseq sequence number        */
        unsigned long acks ;                    /* each bit from right to */
                        /* left indicastes the status of succesive     */
                        /* packets starting with sequence number       */
                        /* given by ack field in mcb_stru              */
        struct node *r_queue ;
        }
```

A.2 Packet Formats

A.2.1 Join.Request Packet

Type	Local_port	Leader_addr	Group_name	ISN

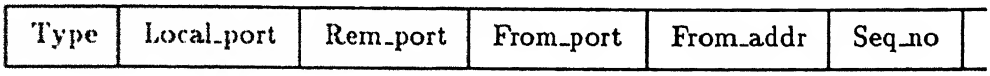
A.2.2 Join.Accept Packet

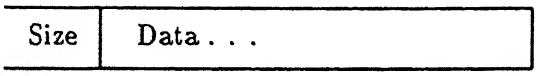
Type	Ser	Local_port	Rem_port	ISN

A.2.3 Join.Deny Packet

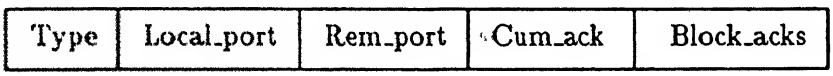
Type	Rem_port	Error

A.2.4 M_cast & Forward Data Packets

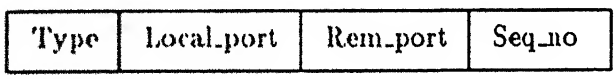




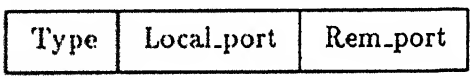
A.2.5 M_ack & F_ack Packets



A.2.6 Quit & Cancel Request Packets



A.2.7 Quit.Confirm & Cancel.Ack Packets



Appendix B

User Manual

Name

create_group, join_group, quit_group, m_send, and m_recv

Interface to the Multicast Transport Protocol.

Library

archive library mlib.a

Synopsis

#include < *mtcp.h* >

int *create_group(*group_name*, *ser*, *soc*, *conn*)
char **group_name*;
int *ser*, *conn*;
struct sockaddr_in *soc*;

int *join_group(*gid*, *mini*, *ser*, *soc*)
struct group_id *gid*;
unsigned long *mini*;
int *ser*;
struct sockaddr_in *soc*;

int quit_group(*mid*)
int **mid*;

int m_send(*mid*, *packet*, *packet_size*)
int **mid*, *packet_size*;
char **packet*;

int m_recv(*mid*, *packet*, *packet_size*)
int **mid*, *packet_size*;
char **packet*;

Parameters

group_name Specifies the name of the group to be created. This is a string and should end with a NULL character.

Ser In create_group it specifies the type of service. It is one of BEST_EFFORT, INTERMEDIATE or RELIABLE or'ed with ECHO_OFF or ECHO_ON. In join_group it specifes only the echo. One of ECHO_OFF or ECHO_ON.

Soc Specifies the socket structure in which the local port number and address to which it is to be bound is specified.

Conn Specifies the number of connections required, before the protocol releases the create_group call.

mini Specifies the internet address of the mini-leader, to which the member wants to connect.

mid This is the pointer, returned by create_group and join_group calls.

packet In m_send, this specifies the address of the packet which is to be sent. In m_recv, this specifies the address at which the incoming packet is to be stored.

packet_size In m_send, this specifies the size of the packet to be multicasted. In m_recv, this specifies the size of the buffer allocated to receive the packet.

gid It is a structure named *group identifer* defined in **mtcp.h** header file. It consists of two fields, *group_name* and *leader_addr*. *Group_name* refers to the group name, in which the member wants to join, and *leader_addr* is internet address of the leader.

Description

Create_group creates a multicast group dynamically. The name of the group should be unique at the host. The process using this function becomes the leader of the group. If ECHO_ON is set, then the packets transmitted by the leader, are also received by this process. This function is blocked by protocol till *conn* (parameter passed to the function) members join the group. If *conn* is zero, the function is not blocked.

With join_group function, a user can join an exiting multicast group. For this, one has to know the name of the group, and the leader's internet address. If ECHO_ON is set, then the packets sent by member are recevied by it. The service provided to the member, is fixed by leader while creating the group. If the mini-leader address passed to the function is non-zero, then the member gets connected to the mini-leader. It receives packets as long as it is in the group.

Quit_group function is used by a member, mini-leader, or leader to quit from the group it has joined. The group does not exit, when the last member quits from the group. Members can communicate with each other, even, when the leader or mini-leader, to which it got connected, had quit the group.

m_send function is used to multicast data to the group, whose identifier is passed to the function. If the group is closed, or if any error occurs, this function fails and returns -1.

m_recv function is used, to recieve the data multicasted to the group. If the group is closed, or if any error occurs, this function fails and returns -1.

Return value

Create_group and join_group returns mulicast indentifier(*mid*) which is an array of three integers. *mid[0]* is the socket identifer and *mid[1]* is message queue indentifer. *Mid[2]* is used as, message type and error value. If *mid[0]* is -1, *mid[2]* represents error value, else *mid[2]* represents the message type(number). If the function is succesfull, *mid[0]* is greater than or equal to zero. If function fails, *mid[0]* is set to -1 and mid[2] is set to the error number.

If quit_group is succesful it returns zero, else it returns -1, and *mid[2]* is set to the error number.

If m_send is succesful, it returns the size of data multicasted. If it fails, it returns -1 and *mid[2]* is set to the error number.

If m_recv is succesful, it returns the size of data recived. If it fails, it returns -1 and *mid[2]* is set to the error number.

Error values

0x01 failed in creating socket at create_group function call

0x02 failed in binding socket at create_group function call

0x03 failed in msgget system call at create_group function call

0x04 failed in sendto system call at create_group function call

0x05 unknown packet received while waiting for CRE_GRO_REP packet

0x06 failed in creating socket at join_group function call

0x07 failed in binding socket at join_group function call

0x08 failed in msgget system call at join_group function call

0x09 failed in sendto system call at join_group function call

0x0a unknown packet received while waiting for JOI_GRO_REP packet

0x0b no free mcb (multicast control block) found

0x0c the group not found at remote address

0x0d free member structure not available at mini-leader

0x0e illegal message number (*mid[2]*), probably modified

0x0f the connection is already closed

0x10 mcb in illegal state, packet ignored

0x11 unknown packet is received in m_recv call

Appendix C

Man Pages

C.1 System Administrator's Manual

Name

dem, cont, send, recv, and timer Processes required for multicast transport protocol.

Synopsis

dem [-d]

cont [-d]

send [-d]

recv [-d]

timer [-d]

Flags

-d causes the process to switch into debug mode. In this mode the process prints the information of the packets it receives, on the standard output.

Description

dem This is the demultiplexing process for the multicast transport protocol. This process demultiplexes the packets received from other hosts to cont, send, and recv processes. Dem is to be executed before the cont, send, and recv processes. This process creates a shared memory segment with key SHM_KEY. SHM_KEY is

defined in header file mtcp.h. If this key is already used by some process, the process exits by returning an error, because it creates the shared memory with IPC_EXCL flag. If this happens then the value of SHM_KEY in header file mtcp.h is to be changed and compiled again. After the shared memory is created a semaphore is created with key SEM_KEY Similar to the SHM_KEY, this key should also be unused, otherwise the process exits. semaphore created is used by send, recv, and cont processes for the mutual exclusive access of shared memory. Dem process receives the packets from other hosts through DEM_PORT. SEM_KEY and DEM_PORT are defined in mtcp.h header file.

cont This is control process for the multicast transport protocol. This process takes care of creating and joining a group. This process is to be executed after the dem process is executed, because cont gets the shared memory identifier and semaphore identifier created by dem process. This process uses CONT_PORT to receive the packets demultiplexed by dem process.

recv This is receive process for the multicast transport protocol. This process takes care of demultiplexing the packets to the users. This process is to be executed after the dem process is executed, because recv gets the shared memory identifier and semaphore identifier created by dem process. This process uses RECV_PORT to receive the packets demultiplexed by dem process.

send This is send process for the multicast transport protocol. This process takes care of multicasting the packets to a group. It also takes care of forwarding data to the members, which depend on this protocol entity. This process is to be executed after the dem process is executed, because send gets the shared memory identifier and semaphore identifier created by dem process. This process uses SEND_PORT to receive the packets demultiplexed by dem process. Send process creates a message queue with key SEND_KEY. This message queue is used by the process to send the acknowledgements for the data received from the functions *m_send* and *quit_group*. If this key is used already by the system the process exits. This key is defined in header file mtcp.h.

timer This is the timer process for the multicast transport protocol. This process takes care of setting the timer for a packet on the request from cont, send, and recv processes. This pacet receives the requests through the TIMER_PORT. TIMER_PORT is defined in the header file mtcp.h.

C.2 Man showmcb

Name

showmcb - Reports the status of multicast control blocks

Description

showmcb gives the status of all non free multicast control blocks (mcb). It gives the information of all the fields in the mcb. If any members are present in the members list of mcb then the function gives the information of this members also. This function gives the index, state, type, no_members, min_members, group_name, service, echo, local_port, leader_addr, mini_addr, mini_port, snext, rnext, pflags, rack, no_packets, no_f_packs and no_m_packs.

index specifies the index of the mcb.

state gives the state of mcb. The state can be ESTABLISHED, JOIN_REQUEST, CLOSED, or CLOSEWAIT.

type specifies the type of protocol entity. It can be LEADER, MINI-LEADER, or MEMBER.

no_members gives the number of members dependent on this protocol entity, to receive data.

min_members gives the number of members still required, to release the create_group function. This is useful only, if the type of mcb is LEADER.

group_name specifies the name of the multicast group.

service specifies the class of service quality provided by this multicast group. It can be BEST_EFFORT, INTERMEDIATE, or RELIABLE.

echo gives the echo status of protocol entity. It can be
ECHO_OFF or ECHO_ON.

local_port specifies the port number on which the user receives data.

leader_add specifies the internet address of the group leader.

mini_addr specifies the internet address of the mini-leader, to which the multicast packets are sent.

mini_port specifies the port number to which the mini-leader is bound.

snext specifies the sequence number that will be used for the next packet to be multicasted.

rnext specifies the sequence number of the packet next to be received from mini-leader(leader).

pflags It is a vector of bits. Bits from right to left gives the status(received or not) of successive packets. The right most bit represents the status of packet, whose sequence number is (rnext + 1).

rack specifies the sequence number of the packet for which the protocol entity is waiting for acknowledgements from its members.

no_packs gives the number of packets in r_queue.

no_f_packs gives the number of packets in f_queue.

no_m_packs gives the number of packets in m_queue.

If any members depending on this protocol entity to receive data are present, then the showmcb function gives the memb_index, memb_port, mem_addr, rseq, and no_packs of the respective member structure in mcb.

memb_index specifies the index of the member structure.

memb_port gives the port number of the member to which it bound.

memb_addr specifies the internet address of the host on which the member is.

rseq gives the sequence number of the packet next to be received from member, that has to be multicasted.

no_packs specifies the number of packets in r_queue of the member structure.